Optimizing Concurrent Tail Inserts into B-Trees

Derek Paulsen

B-Trees are one of the most common data structures in database systems, used to index nearly any kind of data. As the number of cores and memory available in servers running DBMS's have increased, the demand for data structures that can efficiently support concurrent operations has grown exponentially. Simple locking based methods for concurrency control in B-Trees, such as locking the entire tree, or the path to the node that is being read or modified, quickly become a bottleneck in multithreaded workloads in modern servers. To alleviate this bottleneck, many extensions of B-Trees have been proposed that minimize locking (e.g. Latch Coupling [4], B^{link}-Link Trees [5], Optimistic Lock Coupling [1]) or eliminate locking completely (e.g. Bw-Trees [2]) to improve concurrency. While these methods have greatly increased to concurrent throughput over baseline locking algorithms, concurrent tail inserts are still problematic for many algorithms, and have greatly reduced throughput compared to other workloads. In this paper we examine the problem of tail inserts and propose a solution that extends a B-Tree Optimistic Lock Coupling to reduce contention between threads and improve throughput for tail insert workloads.

1. Background

B-Tree indexes have been used in database systems for well over 50 years, dating back to System R [8], where B-Trees where the only index supported. Since then, the number of cores per processor and the size of RAM have increased significantly. Where in the past a B-Tree would need to be stored partially on disk, it is now feasible that for many applications to have the entire B-Tree be stored in main memory. In multithreaded servers with large main memories, disk I/O is no longer a bottleneck, instead the concurrency control mechanisms, which were performant enough for previous architectures, have now become a bottleneck. Many extensions to B-Trees have been proposed and implemented including Latch Coupling [4], B^{link}-Link Trees [5], Optimistic Lock Coupling [1], and Bw-Trees [2]. While these methods have increased the concurrent operation throughput of B-Trees over baseline solutions (e.g. locking the entire tree, or locking the path to the leaf node), concurrent tail inserts into B-Trees still pose a significant problem to these and other algorithms, which can collapse under high contention.

In this paper we will examine the problem of concurrent tail inserts into B-Trees and current algorithms for concurrency control in B-Trees. We will start by reviewing related work on B-Trees. Next we discuss the problem of tail inserts, why it is an issue, and where it might arise. We then propose a solution and give preliminary experimental results to show its effectiveness in dealing the with the problem. Finally, we will discuss our findings and suggest directions for future work.

2 Related Work

2.1 Bw-Trees

Bw-Trees [2] are a form of lock-free B-Tree developed by Microsoft in 2013 which uses delta chains to optimize writes to the tree. The core idea behind Bw-Trees is to allow concurrent writes to occur without locks via the use of delta chains and periodically resolve writes. This approach allows for fast concurrent writes at the cost of read speed as reading requires resolving previous writes to get the current version of the tree. A comprehensive comparison of Bw-Trees with other B-Tree and B-Tree-like indexes was done by [3], in their experiments Bw-Trees were significantly slower than B-Trees with OLC on all workloads, hence we don't consider them any further.

2.2 B-Tree with OLC

Lock coupling has long been used as a concurrency control mechanism on top of B-Trees, however it has the issue that, especially for low contention workloads, it performs a lot of unnecessary locking. To remedy this, optimistic lock coupling (OLC) replaces read locks with version numbers. Instead of acquiring a read lock on a node, the thread reads a version number, which, after reading data in the node, is validated by the thread. If the version number has changed (i.e. the node has been modified) the read procedure restarts from the beginning, else the read continues. In their evaluation, Ziqi Wang et al. [3] found that a B-Tree with OLC was the fastest B-Tree index for concurrent operations, hence we use it as our baseline solution and the B-Tree which we build our solution on top of.

2.3 Other Tree Like Structures

There are many other tree-like data structures that have been developed for use in memory databases such as MassTrees [6] and Adaptive Radix Trees [7] (ART). While Ziqi Wang et al. [3] did show that some of these algorithms (in particular ART), can greatly out perform B-Tree data structures, they are more appropriate to classify as tries or a combination of a trie and a B-Tree. While we find these solutions interesting, we have chosen to focus on optimizing B-Trees for tail inserts, not creating a new data structure. We have chosen this focus because B-Trees are far more common than tries in current database systems, meaning that a solution that adapts B-Trees will likely be much easier to integrate into existing systems than a completely new data structure.

3 Problem

Consider the following workflow,

1. A customer submits an order to a server
2. The server gets the next order id (monotonically increasing) and adds additional information about the order
3. The server then inserts the completed order into the database and then notifies the customer that the order is placed

In this scenario it is likely that the order ids will be indexed since they make an ideal primary key. For a hash index, such a workflow is not an issue since a good hashing algorithm will randomly distribute the keys and thus contention between concurrent modifications should be low. For a B-Tree on the other hand, this workload is very problematic. Since the order ids are monotonically increasing, all concurrent threads will be inserting into the right most leaf of the tree (i.e. a tail insert). For algorithms that take locks to perform updates, the throughput of operations will quickly degrade since all threads need to take the same lock to insert the order. Of course it is possible to randomly select order ids, turning the sequential inserts into a random inserts, however you loose the ability to quickly iterate over the orders in chronological order. Iterating over orders in chronological order maybe very useful when determining which orders to fill first in the event of limited supply, therefore maintaining a monotonically increasing order id in a B-Tree index adds functionality that is not present in other schemes.

4 Solution

As illustrated above, certain workloads are very problematic for some current B-Tree implementations. The key observation is that most algorithms breakdown when there are many concurrent writers on the same leaf of the tree, due to locking for writes. The main goal of our solution is to get rid of locking for writes that would normally be high contention. To do this, we propose a two step solution when handling inserts into the B-Tree. To perform an insert, we first put the (key, value) pair into an unordered buffer. Once the buffer fills, the buffer is replaced with an empty buffer from a preallocated pool and a thread is assigned to flush the buffer into the tree using repeated inserts into the B-Tree. This approach has the distinct advantage that it is completely separate from the underlying B-Tree implementation. That is, this solution can be placed on top of any existing B-Tree algorithm with essentially no modification. For our backing B-Tree implementation we choose to use a B-Tree with Optimistic Lock Coupling since Ziqi Wang et al. [3] showed that it was the best performing B-Tree implementation.

4.1 Ordering of Inserts

While unordered buffers make concurrent inserts easy, it can lead to unintuitive behavior. Say we have a thread *t1* performing inserts into a B-Tree with unique keys. *t1* inserts pair (*k*, *v1*) into buffer *B1*. *B1* then fills and is assigned to another thread to be flushed. *t1* then inserts pair (*k*, *v2*) into buffer *B2*. *B2* then fills and is assigned to another thread to be flushed. If the thread inserting *B1* stalls, *B2* could be flushed before *B1* meaning that it would be possible for the B-Tree to contain (*k*, *v1*) after the workload completes. In fact, we cannot guarantee any sort of consistency for the B-Tree at any point since buffers can be flushed in any order. Clearly this behavior is not acceptable for many applications.

To fix this problem we tag each payload with a monotonically increasing version number after an insert has been placed in an insert buffer or at the time of insert into the B-Tree if the update is performed directly on the B-Tree. We then use this version number to ensure that the most up to date value for a given key is contained in the B-Tree. Specifically, when we insert into the B-Tree, if the key already exists in the tree we check the version numbers of the payloads and store the payload with the greater version number.

4.2 Insert Buffer

Our buffer implementation uses a single lock to synchronize before flushing to the B-Tree, an atomic integer for the next insert position, an atomic integer for the minimum version number, and a fixed size array (currently 1024 elements) to store the inserts. Before a thread is allowed to insert into the buffer, it must first acquire a *shared* lock on the buffer. Once a shared lock on the buffer is acquired, the thread doesn't release it until the buffer is filled. After acquiring a shared lock on the buffer, the thread then attempts to insert into the buffer by atomically fetching and incrementing the current insert position, if the position is valid (i.e. less than the buffer size), the thread inserts the (key, value) pair into the position it fetched and tags it with the next version number. If the position is not valid, the insert fails and the failure is returned to the thread, signaling that the buffer is full. Note that while the getting the insert position is atomic, writing to the buffer is not. Because of the lock synchronization this is not an issue when the buffer is flushed but could potentially be an issue when reading from the buffer, we address this issue below.

```
struct InsertBuffer {
    static constexpr long capacity = 1024;
    std::shared_mutex mu;
    std::atomic<long> pos, min_version;
    std::array< std::pair<K, Versioned<V> >, capacity> buf;
}
```

4.3 Flushing Buffers

If a thread attempts to insert into a buffer and the insert fails, this means that the buffer is full. In this event, the thread first releases its shared lock on the buffer. After releasing the shared lock, the thread attempts to take ownership of the buffer with an atomic compare exchange operation. If the compare exchange operation fails, the thread inserts directly into the B-Tree and returns to the caller. If the compare exchange operation succeeds, the thread then replaces the full buffer with an empty buffer from a preallocated pool and notifies other threads that the new buffer is ready. The thread must then wait for all other threads to complete their inserts. To do this, the current thread takes an *exclusive* lock on the buffer, waiting for other threads to release their shared locks on the buffer. Once the exclusive lock is acquired, the thread then inserts all of the buffered (key, value) pairs into the B-Tree via repeated normal insert operations. Finally, the thread clears the buffer by setting the current insert position to zero and updating the minimum version number of the buffer to the current version number.

```
# version is a shared atomic counter
def insert(key, payload):
    restart:

    current_buffer = get_current_buffer()
    # try locking the buffer for inserts
    if not current_buffer.is_locked_by_this_thread():
        release_shared_locks()
        if not current_bufffer.try_lock_shared():
            goto restart
    # buffer is full
    if current_buffer.insert(key, payload, &version) == FAILURE:
        release_shared_locks()
        # try to take ownership and flush
        if current_buffer.take_ownership():
            replace_insert_buffer()
            # wait for other theads to finish inserts
            current_buffer.lock_exclusive()
            for k,p in current_buffer:
                 btree.insert(k, p)
            # clear buffer, update minimum version, and unlock
            current_buffer.reset(version)
            current_buffer.unlock_exclusive()
        # tag payload
        versioned_payload = Versioned(payload, version++)
        btree.insert(key, versioned_payload)
```

4.4 Reading

Buffering the inserts also adds another complication to consistency aside from the ordering of inserts. Since inserts into the B-Tree are delayed for an indeterminate amount of time, keys may not be immediately visible in the B-Tree after the insert procedure completes. Furthermore, multiple buffers could be getting flushed at any particular time. To solve this problem we allow threads to read directly from the buffers. Before explaining the procedure for reading from the buffers we must first explain the notion of a *valid* buffer read. We say that a read is *valid* if the version number of a payload is greater than the minimum version number of the buffer, less than or equal to the maximum version number of the read, and the minimum version number of the buffer hasn't changed since the start of the read. *Valid* buffer reads are guaranteed to reflect an actual insert into the tree. The first potential issue performing a read is reading stale data since the buffers are recycled. To prevent this, we maintain a minimum version number for each buffer, which, when updated, marks all values in the buffer as invalid. The next potential issue with reading from the buffers is torn writes. As mentioned above, writes into the buffers are *not* atomic, this means that a thread could read an updated key with a payload from another write or read an updated payload with a key from another write. To prevent this, the threads synchronize on the version number, this ensures that if a thread reads data with a version number less than or equal to the maximum version for the write, the key and payload written to the buffer will also be visible. This ensures that torn writes are never returned but it doesn't prevent the buffer from being recycled and overwritten while we are reading it. To prevent this after reading the payload, we compare the current minimum version number to what it was when we started reading the buffer. If the minimum version number has changed, the read maybe invalid, thus we discard the read, however this also means that the buffer has been flushed to the B-Tree, hence the key will be found in the B-Tree.

To perform a read, we first load the current version number (without incrementing). Next, we get the current insert buffer and search it for the key. If the key is found and the read is valid, the payload is retrieved and store. Next, we iterate through the buffer pool, looking for buffers that are currently being flushed. If a buffer is currently being flushed, we search it and if the key is found, and the read is valid, the payload is retrieved and stored. Finally, we search the B-Tree, if the key is found, we retrieve the payload and store it. Finally, we return the payload that has the greatest version number. Searching in this order ensures that any insert, buffered or visible in the tree is always found, since any operation performed which is not found in the buffers will be visible in the B-Tree.

5 Experiments

5.1 Experimental Setup

All experiments where run on single machine with a AMD Ryzen Threadripper 2950X (16 cores/ 32 threads) with 64GB of quad channel running a 2933MHz. The operating system was Pop_OS 18.04 with linux kernel version 5.0.0. All code was with compiled with g++11, with -O3, -ltcmalloc_minimal, -lpthread, and OpenMP. Thread affinity was handled by OpenMP, `OMP_PLACES` was set to `cores` and `OMP_PROC_BIND` was set to `close` for all workloads, these settings have OpenMP to allocate threads as close as possible to the master thread. For each experiment each worker thread fetches the next operation in a shared queue , performs that operation and then fetches the next operation to be done. That is, the operations a particular thread does during the experiment are determined at runtime. This explains the drastic difference of our experimental results when compared with [3], where the authors interleaved the operations by assigning each thread an equal portion of operations a priori in a round-robin fashion (e.g. if there are 2 threads, each thread is assigned every other operation before workload execution begins). We dynamically assign operations for two reasons. First, it is a much more realistic scenario, if the operations were known a priori the tree could be constructed in a much smarter way than sequentially executing the operations. Second, when the operations are assigned a priori for sequential workloads, some worker threads end up falling behind the others which reduces contention and increases throughput. Such an effect hides the effectiveness (or lack there of) of contention regulating mechanisms for our problem setting.

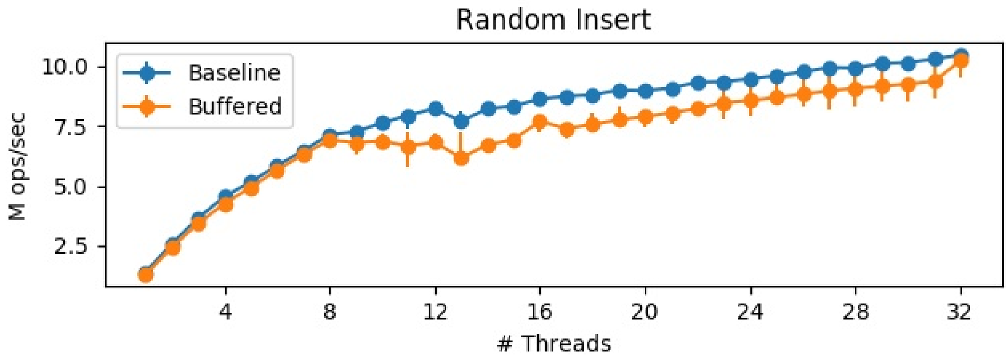
5.2 Workloads

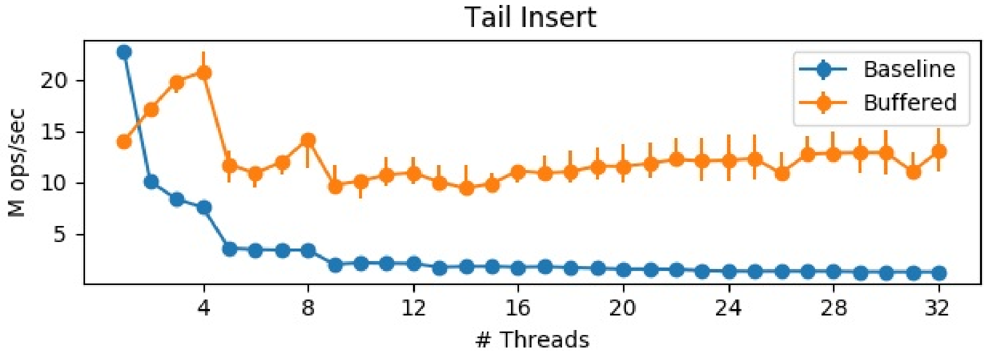
We evaluate our solution on six workloads. Each workload has 8 byte integer keys, and 8 byte integer payloads. We have two insert only work loads and four insert and read workloads. Each has a random and sequential variants. Both insert only workloads are 50M inserts, while the insert and read workloads are 50M inserts and 10M reads and 50M inserts and 50M reads. For the insert and read workloads, the reads a sequenced immediately after the insert of the same key, making it very likely that the read will conflict with the write. We decided to do this to further stress the contention regulation mechanisms of the algorithms tested.

6 Results

All graphs plot the median of 10 runs, with lines showing the range for each experiments. Throughput is measured in millions of operations per second for all graphs.

6.1 Insert Only

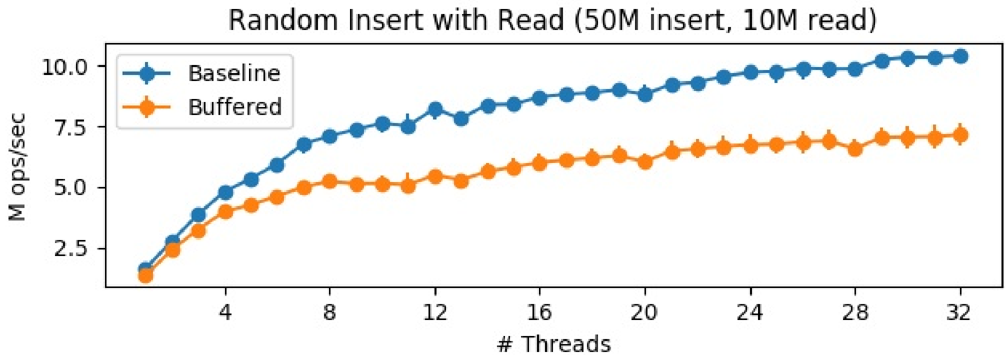


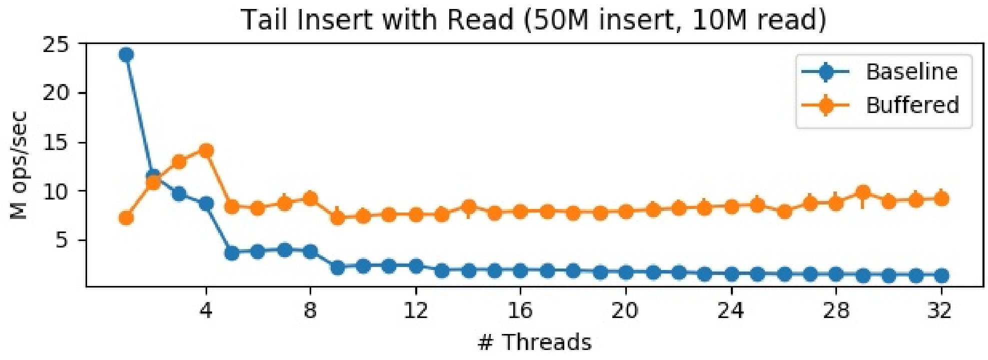


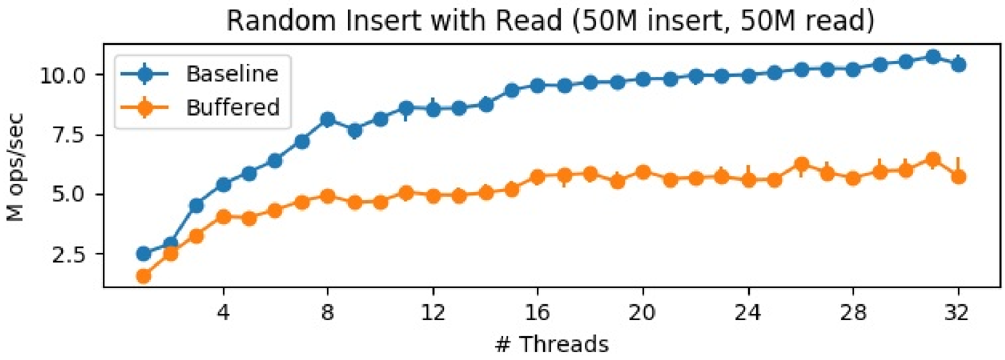
For random inserts, there is relatively low contention, and the memory latency is the bottleneck for the workload. In this case, the buffering mechanism is just overhead, in fact, the buffering mechanism slows the throughput and we see lower throughput once the number of threads increases past 8. Tail inserts, on the other hand, show the effect of the buffer on reducing contention between threads. While the baseline solution completely collapses, the buffer solution levels off at 9.8X the throughput over the baseline.

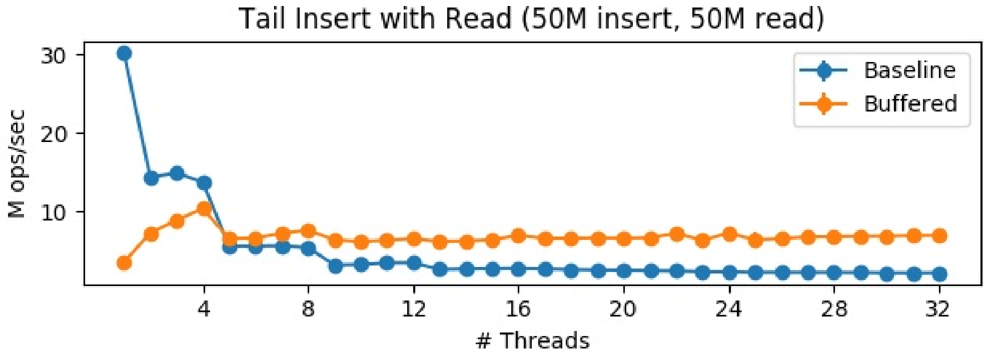
We notice that there are two distinct peaks in the Tail Insert graph at 4 threads and 8 threads. We believe that this is due to the architecture of the Threadripper processor that we used for our experiments. The processor has 16 physical cores (32 logical cores), the physical cores are grouped into cpu complexes (CCXs) of 4 physical cores (8 logical cores). These CCXs are then grouped into two dies which are connected with an inter-die memory interconnect called the 'infinity fabric'. The memory interfaces play a significant role in performance as the intra-CCX memory interface is faster than the intra-die memory interface, which is much faster than the inter-die 'infinity fabric'. Hence up until 4 threads, any cache invalidations are resolved via fast intra-CCX memory interfaces. From 5 to 8 threads, all threads as still collocated on the same die and using the intra-CCX and intra-die interfaces. We see a drop from 4 to 5 threads as a result of the having to rely on the intra-die interface. Similarly we see a drop from 8 to 9 cores since we are then having to rely on the inter-die 'infinity fabric'. Finally, we notice that the same pattern holds for all tail insert workloads where the worker threads are all trying to access and modify the same memory regions concurrently.

6.2 Insert and Read









When reads are added into the workload, the picture stays roughly the same, although the buffered solution does worse on both random and tail inserts. This is due to the increased cost of reads since all non-empty buffers must be searched along with the B-Tree itself. When the workload is 50% reads, the effect is very obvious as our throughput on the tail insert workload levels off at only 3.1X the throughput of the baseline.

7 Discussion

7.1 Optimize Flushing Buffers

Astute readers will likely have noticed that buffered writes could be flushed to the tree in a more efficient way. Specifically, one could traverse the tree and insert some or all of the buffered writes into the appropriate node(s), without having to traverse the tree from root to leaf for each write. This would likely greatly increase the throughput of tail inserts and could possibly help with random inserts by avoiding some cache misses. This optimization, while simple to state, is not easy to implement, hence we defer it to future research to explore implementations of bulk writing optimizations.

7.2 Other operations

Our evaluation only considers insert and read operations, of course there are more operations that a B-Tree is used for in many systems. In particular, future work should examine delete and range scan operations. For the later we note that our buffering implementation can be generalized from storing (key, value) pairs to storing (key, value, operation) triples, where operation could be insert, delete, or any other operation that might be performed on data in the B-Tree. Deletes are particularly interesting for the buffering strategy since it may allow for completely omitting operations on the B-Tree if there is an insert and delete operation on the same key in a single buffer.

7.3 Contention Regulation vs. Workload

As mentioned above, on low contention workloads, the buffering mechanism we propose is essentially just overhead. This points to a fundamental idea in concurrent algorithms, namely, matching the algorithm to the workload. For example, read-only workloads could run extremely quickly with a single global shared lock, since there is no contention between threads, any contention regulation or concurrency control mechanisms are just overhead that will slow down the workload. In contrast, high contention workloads in the worst case can completely collapse with the wrong algorithm becoming significantly worse than a single threaded solution. We think the solution we present in this paper is another data point in the space of this trade-off. Future work can address tuning the concurrency control mechanism to the workload dynamically to give more consistent performance across a variety of workloads.

8 Conclusion

B-Trees are one of the most popular indexes used in database systems and will likely continue to be for a long time to come. Their relative simplicity and efficiency make them appealing indexing solution for any data that has some meaningful order to it. Additionally, databases are only going to become more and more parallel. As the amount of parallelism in databases increases, the need for efficient concurrency control algorithms for B-Trees will continue to increase. Here we present an algorithm to optimize tail inserts into a B-Tree using a relatively simple buffering solution. Finally, we recognize that our solution has opportunities for optimization and that future research could certainly make improvements upon the algorithm presented in this paper.

Sources

[1] Leis, Viktor et al. “Optimistic Lock Coupling: A Scalable and Efficient General-Purpose Synchronization Method.” IEEE Data Eng. Bull. 42 (2019): 73-84.

[2] J. J. Levandoski, D. B. Lomet and S. Sengupta, "The Bw-Tree: A B-tree for new hardware platforms," 2013 IEEE 29th International Conference on Data Engineering (ICDE), 2013, pp. 302-313, doi: 10.1109/ICDE.2013.6544834.

[3] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G. Andersen. 2018. Building a Bw-Tree Takes More Than Just Buzz Words. In *Proceedings of the 2018 International Conference on Management of Data* (*SIGMOD '18*). Association for Computing Machinery, New York, NY, USA, 473–488. DOI:https://doi.org/10.1145/3183713.3196895

[4] Goetz Graefe. 2011. Modern B-Tree Techniques. *Found. Trends databases* 3, 4 (April 2011), 203–402. DOI:https://doi.org/10.1561/1900000028

[5] Philip L. Lehman and s. Bing Yao. 1981. Efficient locking for concurrent operations on B-trees. *ACM Trans. Database Syst.* 6, 4 (Dec. 1981), 650–670. DOI:https://doi.org/10.1145/319628.319663

[6] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache craftiness for fast multicore key-value storage. In Proceedings of the 7th ACM european conference on Computer Systems (EuroSys '12). Association for Computing Machinery, New York, NY, USA, 183–196. DOI:https://doi.org/10.1145/2168836.2168855

[7] V. Leis, A. Kemper and T. Neumann, "The adaptive radix tree: ARTful indexing

for main-memory databases," 2013 IEEE 29th International Conference on Data Engineering (ICDE), 2013, pp. 38-49, doi: 10.1109/ICDE.2013.6544812.

[8] Astrahan, M., M. Blasgen, D. Chamberlin, K. Eswaran, J. Gray, Patricia P. Griffiths, W. F. King, R. Lorie, P. McJones, J. W. Mehl, G. R. Putzolu, I. Traiger, B. W. Wade and V. Watson. “System R: relational approach to database management.” ACM Trans. Database Syst. 1 (1976): 97-137.